.NET Full Stack Development Program

—

Day 3 OOP

Outline

- Class and Object
- Object-oriented programming (OOP)
  - Encapsulation
  - Inheritance
  - Polymorphism
  - Abstract
- Object class

OOP Intro

- OOP stands for Object-Oriented Programming
- The goal of OOP is to group up some data and operations as a single unit called an “Object”
- Object is a small unit in the program that represents a real-world entity

Class

- C# is an object-oriented programming language, everything in C# is associated with classes and objects.
- A class is a “blueprint” for creating objects.

Class

- To create a class we need to use the keyword – ***class***
- Normally we will create a class within a namespace

```
class Employee // Please use Pascal Case to name your class
{
  // states - member fields

  // behaviors - member methods

  // constructors
}
```

Class Members

- Fields and methods inside classes are often referred to as “Class Members”.
- The variables inside a class are called fields.
- Methods are used to perform certain actions.

```
4 references
public enum Gender
{
    Man,
    Woman
}
9 references
public class Employee
{
    //Fields - to describe the state of Employee
    public string _name;
    public int _age;
    public Gender _gender;
    public Employee _manager;

    //Methods - to describe the behavior of Employee
    1 reference
    public void SayHello()
    {
        Console.WriteLine("Hello");
    }


}
```

Constructors

- A constructor is a **special method** that is used to initialize objects.
- It can be used to set initial values for fields

```
public class Employee
{
    //Fields - to describe the state of Employee
    public string _name;
    public int _age;
    public Gender _gender;
    public Employee _manager;

    //default constructor
    1 reference
    public Employee() { }

    //a constructor that can initialize name, age and gender fields
    2 references
    public Employee(string name, int age, Gender gender)
    {
        _name = name;
        _age = age;
        _gender = gender;
    }

    //Methods - to describe the behavior of Employee
    1 reference
    public void SayHello()
    {
        Console.WriteLine("Hello");
    }
}
```

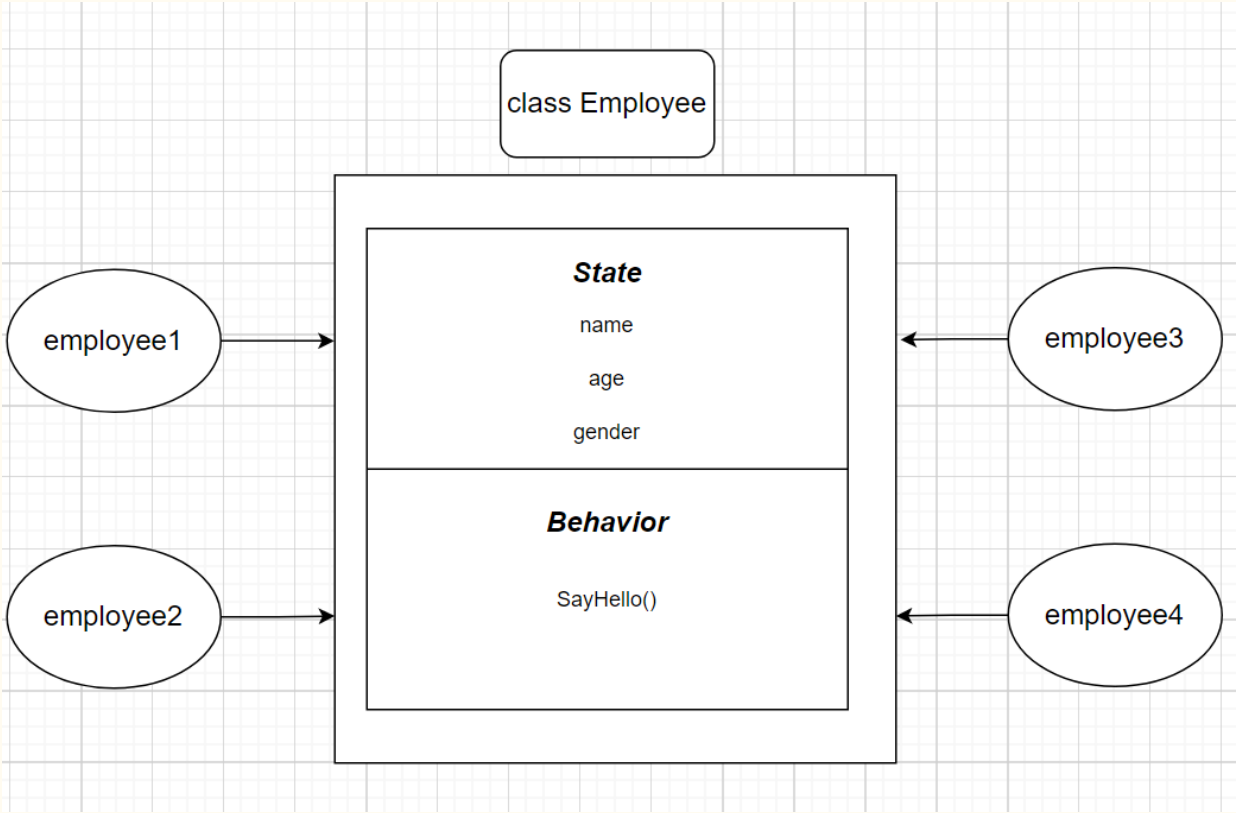
Object

- An **object** is the instance of the class, it has states and behaviors as well.
- The state of an object is stored in fields (variables), while methods (functions) display the object's behavior

- To create an object of a class, specify the class name. followed by the object name, and use the keyword new.

```
static void Main(string[] args)
{
    //use default constructor to instanciate an object called rose
    Employee rose = new Employee();
    rose._name = "Rose";
    rose._gender = Gender.Woman;
    rose._age = 22;
    rose.SayHello();
    //use the parameterized constructor to instanciate an object of Employee class called jack
    Employee jack = new Employee("Jack",23,Gender.Man);
    Console.WriteLine("Employee name: {0}, age: {1}, gender: {2}", jack._name, jack._age, jack._gender);
```

Object vs Class



Static

- declare a static member, which belongs to the **class** itself rather than to a specific object
  - Static
    - Class
    - Compile time
  - Non-static
    - Object
    - Runtime

Static Variable

- Static variable
  - Static fields are stored outside of the object
  - Static fields are common to all objects of a class

Class variable vs. Instance variable

	Class variable	Instance variable
Declaration	declared with ***static*** keyword	declared without static keyword
Storage	stored in **class's memory**	stored in the respective object
Representation	represents **common data** to all objects of the class	represents data related to the respective object
Accessibility	can be accessed by **class only**	can be accessed by objects

Static Method

- Static method
  - static method vs. instance method

	static Method	Non-*static* Method
Access instance variables?	no	yes
Access *static* class variables?	yes	yes
Call *static* class methods?	yes	yes
Call non-*static* instance methods?	no	yes
Use the object reference *this*?	no	yes

Static Class

- Static class
  - If the static keyword is applied to a class, all the members of the class must be static

This keyword

This keyword refers to the current object;

Usage:

- ***this*** is often used to differentiate between the constructor parameters and class fields if they both have the same name.

- ***this*** is also used as a modifier of the first parameter of an extension method.

```
class Employee
{
    //Fields - to describe the state of Employee
    public string name;
    public int age;
    public Gender gender;
    public Employee manager;

    //default constructor
    1 reference
    public Employee() { }

    //a constructor that can initialize name, age and gender fields
    1 reference
    public Employee(string name, int age, Gender gender)
    {
        this.name = name;
        this.age = age;
        this.gender = gender;
    }
```

Extension Method

- It is a new feature that has been added in C# 3.0 which allows us to add new methods into a non-static class without editing the code of the class.

- Extension methods must be defined only under the static class.

- Syntax:

```
[access_modifier] static [return_type] [function_name]
(this [extension_class_name] [binding param], [param list])
{
//your extension methods here
}
```

Extension Method

```
7 references
class Employee
{
    //Fields - to describe the state of Employee
    public string name;
    public int age;
    public Gender gender;
    public Employee manager;

    //default constructor
    1 reference
    public Employee() { }

    //a constructor that can initialize name, age and gender fields
    1 reference
    public Employee(string name, int age, Gender gender)
    {
        this.name = name;
        this.age = age;
        this.gender = gender;
    }


    //Methods - to describe the behavior of Employee
    1 reference
    public void SayHello()
    {
        Console.WriteLine("Hello");
    }


}
```

```
//Extension class
0 references
static class ExtensionForEmployee
{
    1 reference
    public static void OpenTheDoor(this Employee value)
    {
        Console.WriteLine("Door Opened...");
    }


    1 reference
    public static void MorningGreeting(this Employee value, string empName)
    {
        Console.WriteLine($"Good Morning, {empName}!");
    }
}
```

```
//Extension Method
Employee han = new Employee("Han", 32, Gender.Man);
han.OpenTheDoor();
han.MorningGreeting("Mark");
```

Object-Oriented Programming

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Encapsulation

- Encapsulation is hiding information.

- How
  - Declare fields/variables as private.
  - Provide public get and set methods, to access and update the value of a private field.

- Why
  - Flexibility — Internal logic changes won’t affect the caller of the method
  - Reusability — Encapsulated code can be used by different callers
  - Maintainability — Operations on encapsulated unit won’t affect other parts

Property

- Property is an example of Encapsulation, we can use property to access and update the value of a private field.

```
0 references
internal class People
{
    private string name; // this is a private field

    0 references
    public string Name // this is a public property
    {
        get { return name; } // get method to return the people name
        set { name = value; }// set method to change the people name to an assigned value
    }
}
```

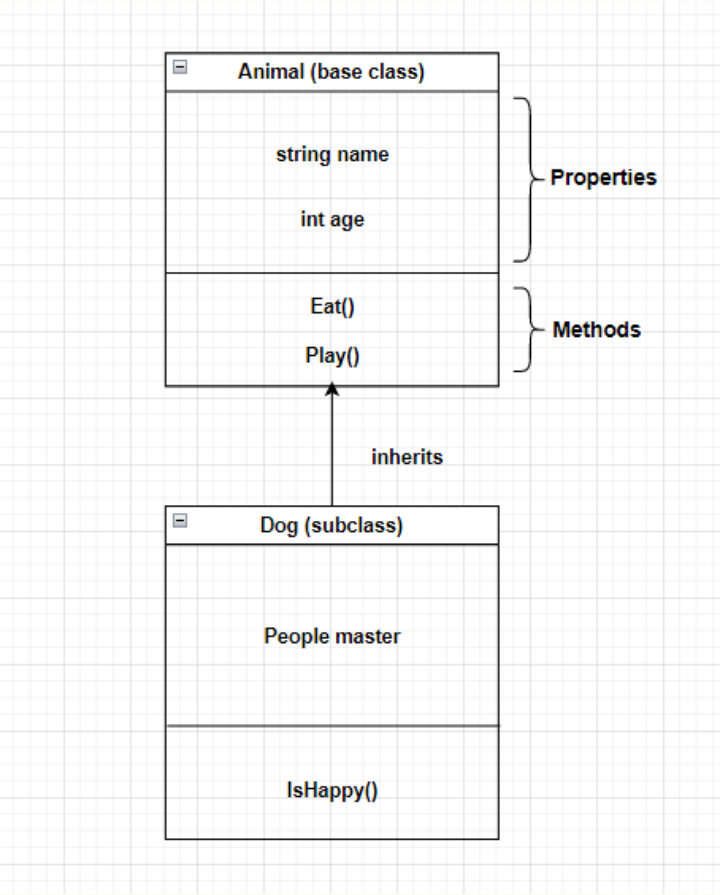
Automatic Property

```
2 references
internal class People
{
    //automatic property
    2 references
    public string Name { get; set; }

    // we can also make the property to get only or set only to meet our needs
    2 references
    public int WorkingHoursPerDay { get; } = 8;
```

```
//Property Example
People tommy = new People();
tommy.Name = "Tommy"; // use property to set the name of People
Console.WriteLine(tommy.Name); // use property to get the name of People
tommy.WorkingHoursPerDay = 15; // this property is read only, we cannot modify it
Console.WriteLine(tommy.WorkingHoursPerDay);
```

Inheritance

- Is the ability to derive something specific from something generic.
- A class can inherit the features of another class and add its own modification.
- The parent class is the base class and the child class is known as the subclass or derived class.
- A subclass inherits all the properties and methods of the base class.
- Aids in the reuse of code.



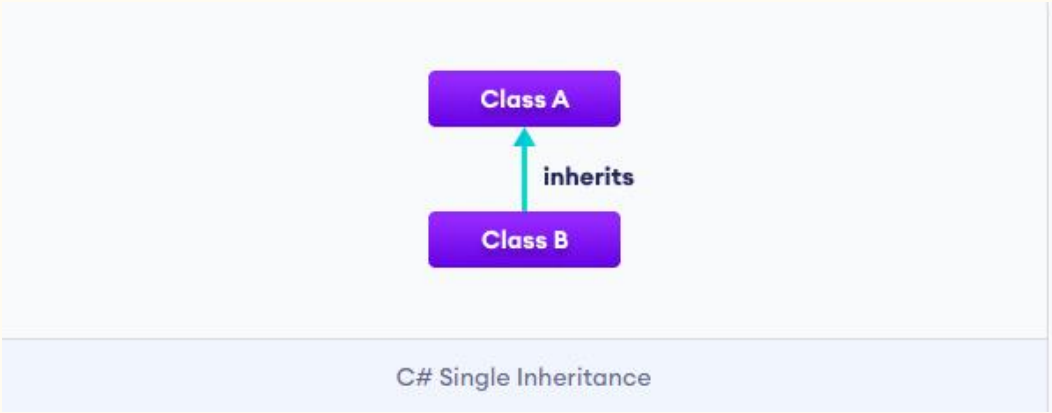
Inheritance

- Syntax
  - ***class* SubClass : BaseClass**
  - It declares that SubClass inherits the base class BaseClass
- IS-A relationship
  - Inheritance implies that there is an IS-A relationship between subclass and base class
  - Eg. Dog is an Animal; Car is a Vehicle; Triangle is a Shape

Type of Inheritance

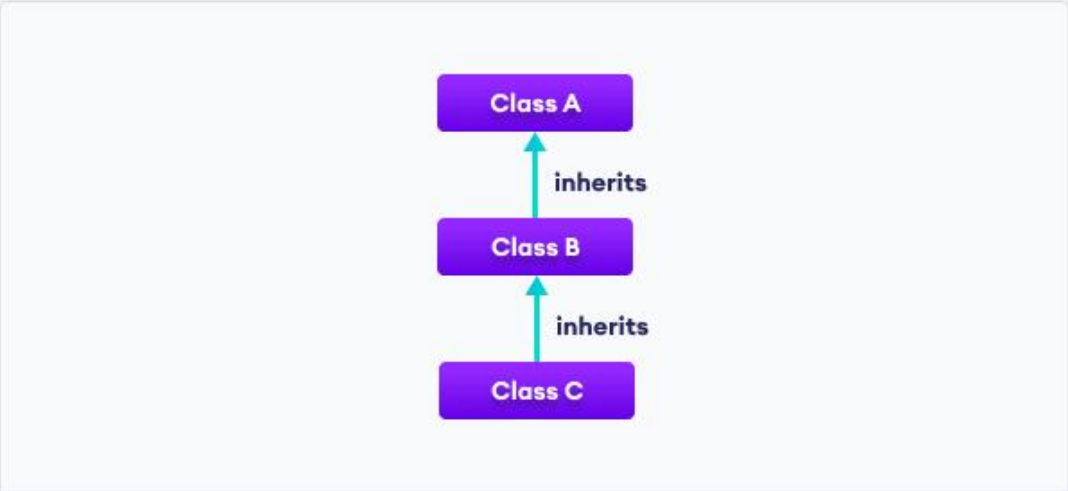
- Single
- Multiple
- Multilevel
- Hierarchical
- Hybrid

Single Inheritance



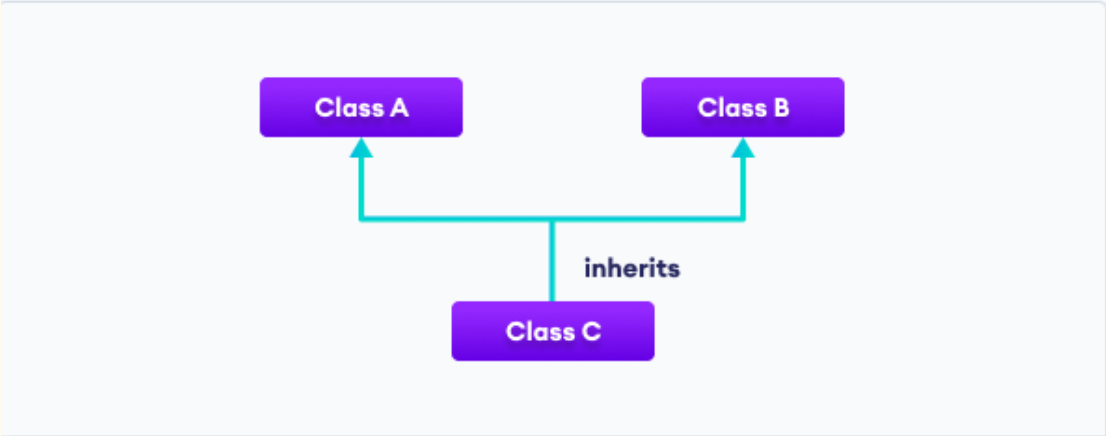
C# Single Inheritance

Multilevel Inheritance

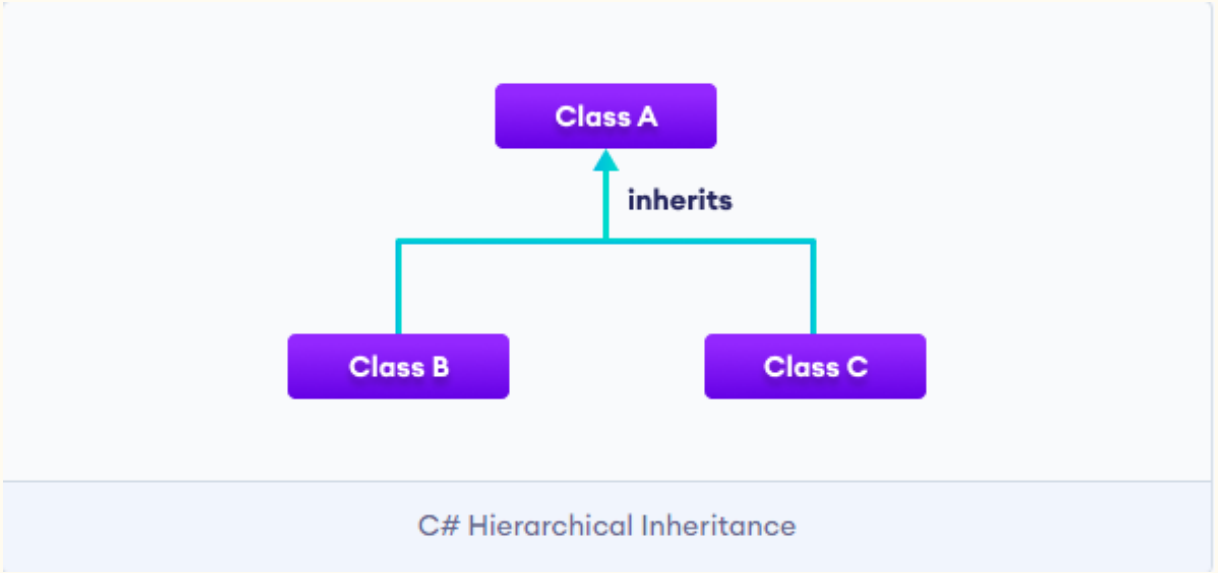


C# Multilevel Inheritance

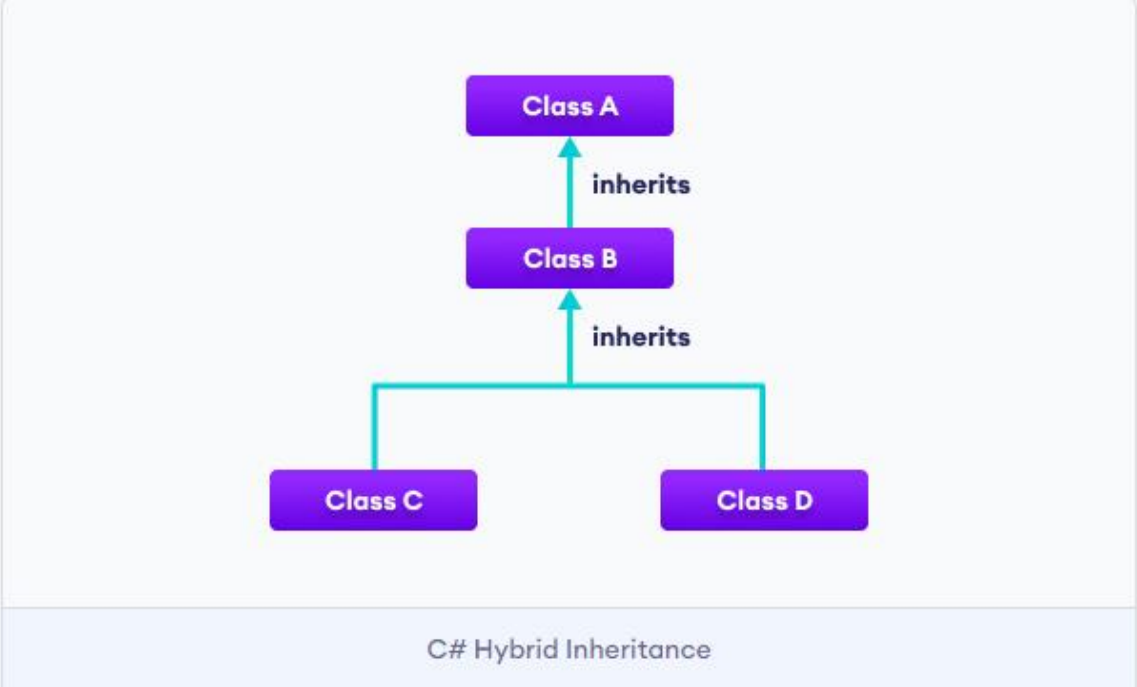
Multiple Inheritance



Hierarchical Inheritance



Hybrid Inheritance



C# Hybrid Inheritance

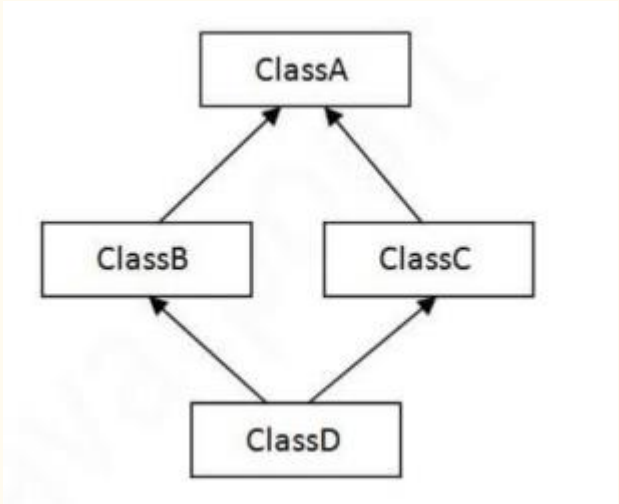
Inheritance in C#

- C# supports the following inheritance:
  - Single
  - Multi-level
  - Hierarchical

- How about Multiple and Hybrid?

Diamond Problem

- An ambiguity that can arise as a consequence of allowing multiple inheritance



Is there any way to resolve the ambiguity caused by the diamond problem?

Interface

- Like a class, an interface can have methods, but the methods declared in the interface are by default abstract (only method signature, no method body)

    - Interfaces specify what a class or a struct must do rather than how to do.
    - It defines a contract, any class or struct that implements that contract must provide an implementation of the members defined in the interface.
    - If a class implements an interface and does not provide method bodies for all functions specified in the interface, then the class must be declared abstract.

Interface

- Syntax
  - public *interface* IShape {}
  - public class Triangle **:** IShape {}

- C# allows **multiple** inheritance of interface

```
1 reference
interface IShape
{
    1 reference
    void PrintShape(); // abstract by default
}
0 references
class Triangle : IShape
{
    //must implement the abstract method in IShape interface
    1 reference
    public void PrintShape()
    {
        Console.WriteLine("Triangle");
    }
}
```

```
interface ISpeak
{
    2 references
    void SayHello(); // abstract by default

}
2 references
interface ISay
{
    2 references
    void SayHello(); //abstract by default

}
2 references
class Person : ISpeak, ISay
{
    4 references
    public void SayHello()
    {
        Console.WriteLine("Hello") ;
    }
}
```

Aggregation

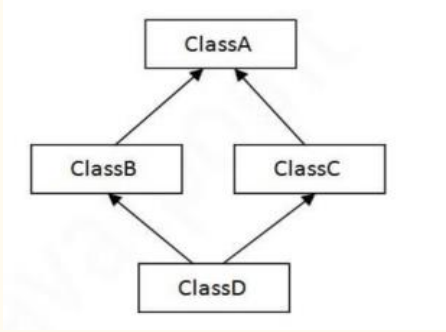
- If a class has an entity reference, it is known as Aggregation
- Aggregation represents HAS-A relationship

```
internal class Employee
{
    0 references
    public int Id { get; set; }
    0 references
    public string Name { get; set; }
    0 references
    public Address Address { get; set; } // Employee Has-An Address


}

1 reference
internal class Address
{
    0 references
    public string Street { get; set; }
    0 references
    public string City { get; set; }
    0 references
    public int ZipCode { get; set; }
}
```

Aggregation

- Code reuse is also best achieved by aggregation when there is no is-a relationship.
- If we don’t have the Address class
  - Have to add all street, city information in employee
  - If we introduce an object called Company later which has the address too, we will have to add same attributes to Company object again

Aggregation



- How does Aggregation solve the diamond problem?
  - Instead of inheriting class B and C, introducing the B and C as the instance variables in class D
  - Thus, we can use b.DoSomething() or c.DoSomething() to eliminate the ambiguity

Inheritance vs. Aggregation

- Inheritance should be used only if the relationship Is-A is maintained throughout the lifetime of the objects involved; otherwise, aggregation is the best choice.

Polymorphism

- **Polymorphism** in c# is a concept by which we can perform a single action in different ways.
  - The word "poly" means many and "morphs" means forms. So polymorphism means many forms.
- There are two types of polymorphism in c#
  - **Compile time** polymorphism (achieved by **method overloading**)
  - **Runtime** polymorphism (achieved by **method overriding**)

Compile time vs. Runtime

- Compile time — the instance where the code you entered is converted to executable

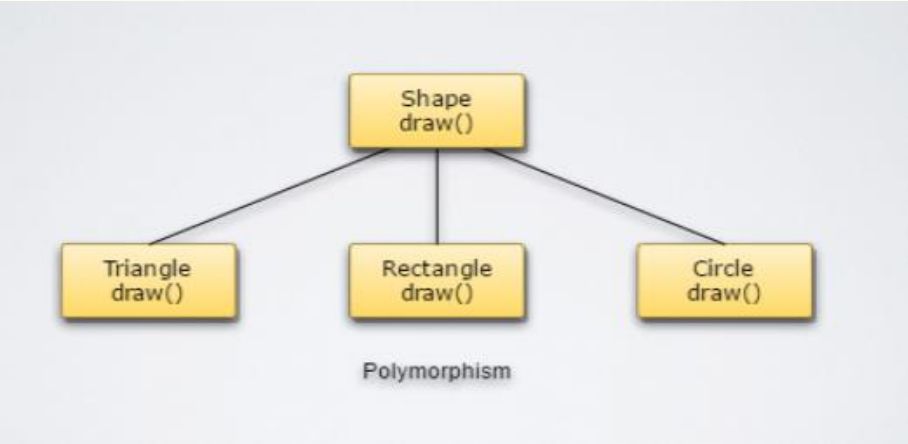
- Runtime — the instance where the executable is running

Runtime Polymorphism

- The form is determined at runtime
- Method **overriding**
  - a method in a subclass has the **same name, return type, and parameters** as a method in its base class, then the method in the subclass is said to override the method in the base class
  - When an overridden method is called through the subclass object, it will always refer to the version of the method defined by the subclass. The base class version of the method is hidden.

Virtual and Override Keywords

- The ***virtual*** keyword is used to specify the virtual method in the base class, and the method with the same signature that needs to be overridden in the derived class is preceded by ***override*** keyword.
    - By default, methods are **non-virtual**. You cannot override a non-virtual method.
    - You cannot use the ***virtual*** keyword with the ***static***, ***private***, ***abstract***, or ***override*** keywords




```
//base class
3 references
public class Shape
{
    //virtual method
    3 references
    public virtual void Draw()
    {
        Console.WriteLine("Performing base class drawing tasks!");
    }
}
```

```
public class Circle : Shape
{
    //override the virtual method in base class
    1 reference
    public override void Draw()
    {
        Console.WriteLine("Drawing a circle");
    }
}
0 references
public class Rectangle : Shape
{
    //override the virtual method in base class
    1 reference
    public override void Draw()
    {
        Console.WriteLine("Drawing a rectangle");
    }
}
0 references
public class Triangle : Shape
{
    //override the virtual method in base class
    1 reference
    public override void Draw()
    {
        Console.WriteLine("Drawing a triangle");
    }
}
```

Compile Time Polymorphism

- The form is determined at compile time
- Method **overloading**
  - a class has multiple methods that have the same name but different parameters
    - Different number of parameters
    - Different data types of parameters
  - Method overloading increases the readability of the program

Compile Time Polymorphism

```
public class Program
{
    0 references
    public static void PrintArea(int x, int y)
    {
        Console.WriteLine(x * y);
    }
    0 references
    public static void PrintArea(int x)
    {
        Console.WriteLine(x * x);
    }
    0 references
    public static void PrintArea(int x, double y)
    {
        Console.WriteLine(x * y);
    }
    0 references
    public static void PrintArea(double x)
    {
        Console.WriteLine(x * x);
    }
```

Overriding vs. Overloading

Overriding vs Overloading in C#	
Overriding in C# is to provide a specific implementation in a derived class method for a method already existing in the base class.	Overloading in C# is to create multiple methods with the same name with different implementations.
Parameters	
In C# Overriding, the methods have the same name, same parameter types and a same number of parameters.	In C# Overloading, the methods have the same name but a different number of parameters or a different type of parameters.
Occurrence	
In C#, overriding occurs within the base class and the derived class.	In C#, overloading occurs within the same class.
Binding Time	
The binding of the overridden method call to its definition happens at runtime.	The binding of the overloaded method call to its definition happens at compile time.
Synonyms	
Overriding is called as **runtime polymorphism**, **dynamic polymorphism** or **late binding**.	Overloading is called as **compile time polymorphism**, **static polymorphism** or **early binding**.

If I want to use the implementation in the base class, how to refer to the base class?

Base

- The **base** keyword is used to access members of the base class from within a derived class
- Usage
  - Call a method on the base class that has been overridden by another method.
  - Use base() to specify which base-class constructor should be called when creating instances of the derived class.

C#

```
public class Person
{
    protected string ssn = "444-55-6666";
    protected string name = "John L. Malgraine";

    public virtual void GetInfo()
    {
        Console.WriteLine("Name: {0}", name);
        Console.WriteLine("SSN: {0}", ssn);
    }
}
class Employee : Person
{
    public string id = "ABC567EFG";
    public override void GetInfo()
    {
        // Calling the base class GetInfo method:
        base.GetInfo();
        Console.WriteLine("Employee ID: {0}", id);
    }
}

class TestClass
{
    static void Main()
    {
        Employee E = new Employee();
        E.GetInfo();
    }
}
/*
Output
Name: John L. Malgraine
SSN: 444-55-6666
Employee ID: ABC567EFG
*/
```

C#

```
public class BaseClass
{
    int num;

    public BaseClass()
    {
        Console.WriteLine("in BaseClass()");
    }

    public BaseClass(int i)
    {
        num = i;
        Console.WriteLine("in BaseClass(int i)");
    }

    public int GetNum()
    {
        return num;
    }
}

public class DerivedClass : BaseClass
{
    // This constructor will call BaseClass.BaseClass()
    public DerivedClass() : base()
    {
    }

    // This constructor will call BaseClass.BaseClass(int i)
    public DerivedClass(int i) : base(i)
    {
    }

    static void Main()
    {
        DerivedClass md = new DerivedClass();
        DerivedClass md1 = new DerivedClass(1);
    }
}
/*
Output:
in BaseClass()
in BaseClass(int i)
*/
```

How can we prevent one method from being overridden?

Sealed

- Method – prevent overriding specific virtual methods

```
class X
{
    protected virtual void F() { Console.WriteLine("X.F"); }
    protected virtual void F2() { Console.WriteLine("X.F2"); }
}

class Y : X
{
    sealed protected override void F() { Console.WriteLine("Y.F"); }
    protected override void F2() { Console.WriteLine("Y.F2"); }
}

class Z : Y
{
    // Attempting to override F causes compiler error CS0239.
    // protected override void F() { Console.WriteLine("Z.F"); }

    // Overriding F2 is allowed.
    protected override void F2() { Console.WriteLine("Z.F2"); }
}
```

- Class – prevent inheritance
  - sealed class SealedClass {}

Abstraction

- **Hiding internal details** and showing functionality
  - eg. phone call, we don't know the internal processing
- Abstraction is achieved by creating either **Abstract Classes** or **Interfaces** on top of your class
- Why
  - Hide the unnecessary things from user so providing easiness.
  - Hiding the internal implementation of software so providing security

```
//To declare an abstract class
1 reference
abstract class Animal
{
    //Abstract method
    2 references
    public abstract void AnimalSound();

    // normal method
    1 reference
    public void Sleep()
    {
        Console.WriteLine("Zzz");
    }
}

// subclasses should override the abstract methods
2 references
class Dog : Animal
{
    2 references
    public override void AnimalSound()
    {
        Console.WriteLine("Woof, Woof"); ;
    }
}
```

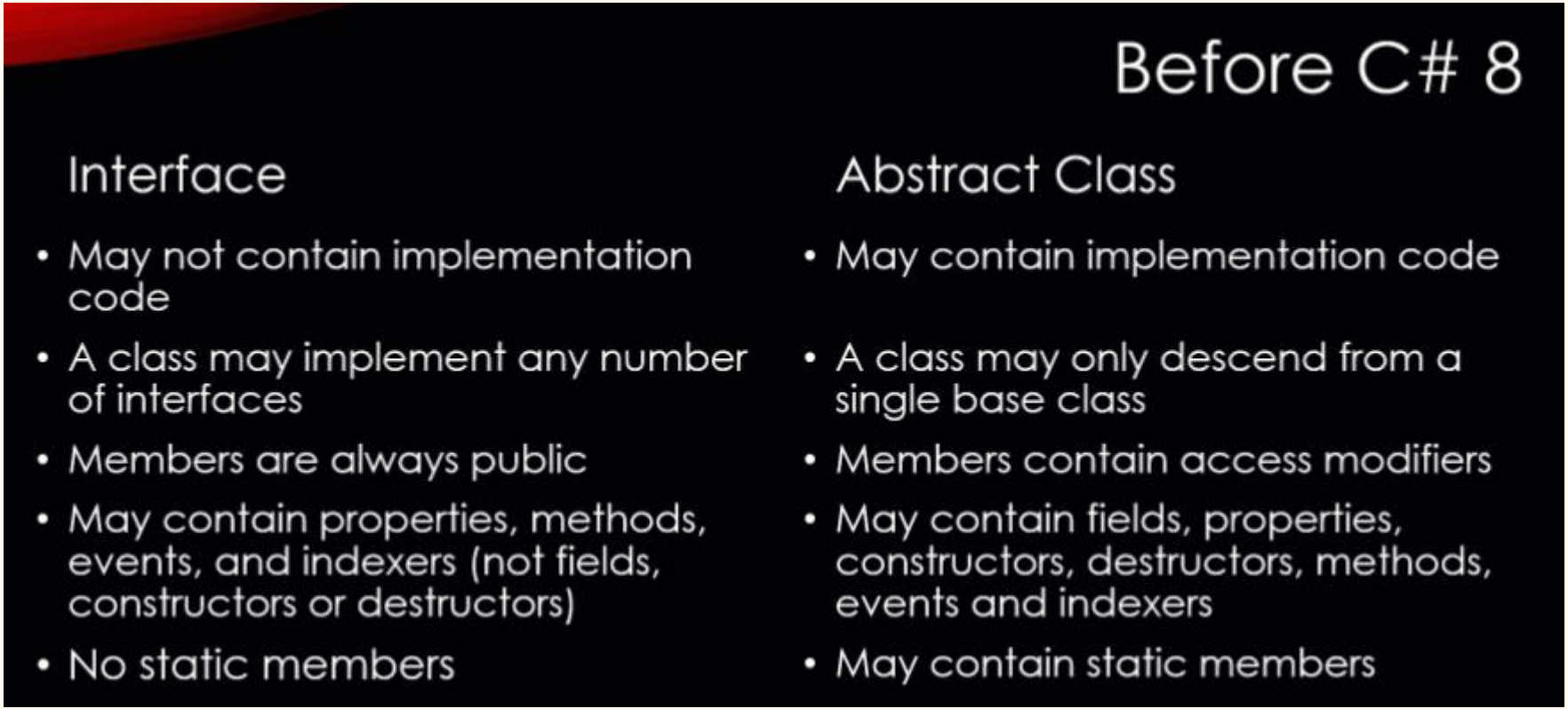
Abstract Class

- Abstract classes are classes with a generic concept, not related to a specific class.
- Abstract classes define **partial behavior** and leave the rest for the subclasses to provide
- Contain zero or more abstract methods.
- Abstract method contains no implementation (like the method in Interface)
- Abstract classes **cannot be instantiated,** but they can have a reference variable
- If the subclasses do not override the abstract methods of the abstract class, then it is mandatory for the subclasses to tag themselves as abstract.

Why Abstract Class

- To force the same name and signature pattern in all the subclasses
- To have the flexibility to code these methods with their own specific requirements
- To prevent accidental initialization
- To define common attributes or methods

Abstract Class vs. Interface



Abstract Class vs. Interface

After C# 8

Interface

- ~~May not contain implementation code~~
- A class may implement any number of interfaces
- ~~Members are always public~~
- ~~May contain properties, methods, events, and indexers (not fields, constructors or destructors)~~
- ~~No static members~~

Abstract Class

- May contain implementation code
- A class may only descend from a single base class
- Members contain access modifiers
- May contain fields, properties, constructors, destructors, methods, events and indexers
- May contain static members

Object Class

Object Class in C#

All types in the .NET type system **implicitly inherit** from **Object** or a type derived from it. The common functionality of **Object** is available to **any type**.

- The Object class is beneficial if you want to refer any object whose type you don't know. Notice that the parent class reference variable can refer to the child class object, known as **upcasting**

Methods of Object Class

Equals(Object)	Determines whether the specified object is equal to the current object.
Equals(Object, Object)	Determines whether the specified object instances are considered equal.
Finalize()	Allows an object to try to free resources and perform other cleanup operations before it is reclaimed by garbage collection.
GetHashCode()	Serves as the default hash function.
GetType()	Gets the Type of the current instance.
MemberwiseClone()	Creates a shallow copy of the current Object.
ReferenceEquals(Object, Object)	Determines whether the specified Object instances are the same instance.
ToString()	Returns a string that represents the current object.

Object Casting

```
2 references
class Grandparent
{
    0 references
    public string Name { get; set; } = "Grandparent";
}

6 references
class Parent : Grandparent
{
    0 references
    public bool IsHealthy { get; set; } = true;
    0 references
    public List<Child> Children { get; set;} = new List<Child>();
}

5 references
class Child : Parent
{
    0 references
    public Parent TheParent { get; set; }
}
```

```
0 references
static void Main(string[] args)
{
    Grandparent grandparent = new Parent(); //up cast
    Parent parent1 = grandparent;
    Parent parent2 = (Parent)grandparent; // down cast
    Child child1 = grandparent;
    Child child2 = (Child)grandparent; // down cast


}
```

Recap

- Class and Object
- Object-oriented programming (OOP)
- Encapsulation
  - Encapsulation
  - Inheritance
  - Polymorphism
  - Abstraction
- Object class

Question?